FIFO and LIFO incognito

Kees van der Laan

February 1992

Abstract

FIFO, first-in-first-out, and LIFO, last-in-last-out, are well-known techniques for handling sequences. In TEX macro writing they are abundant but are not easily recognized as such. TEX templates for FIFO and LIFO are given and their use illustrated.

1 Introduction

It started with the programming of the Tower of Hanoi in TEX, see van der Laan (1992). For printing each tower the general FIFO — First-In-First-Out[1] — approach was considered.[2] In literature (and courseware) the programming of these kind of things is done differently by each author, inhibiting intelligibility. In pursuit of Wirth (1976) TEX templates for the FIFO (and LIFO) paradigma will hopefully improve the situation.

2 FIFO

FIFO can be TEXed as template via[3]

```
\def\bfifo#1#2\efifo{\process{#1}
\ifx\empty#2\empty
\else\bfifo#2\efifo\fi
}%end \bfifo...\efifo
```

Printing of a tower can be done via

```
\def\process#1{\kern.2ex\hbox to3ex{%
\hss\vrule width#1ex height1ex\hss}}
\vbox{\offinterlineskip\bfifo12\efifo}
```

The `\bfifo...\efifo` macro is a basic one. It allows to procede along a list and to apply a (user) specified process to each list element. By this approach the programming of walking through a list is separated from the various processes to be applied to the elements. Fundamental![4]

The recursion will be terminated if #2 is empty.[5] One can circumvent the building up of `\fi`'s via[6]

```
\def\bfifo#1#2\efifo{\process{#1}
   \ifx\empty#2\empty
   \else\def\aux{\bfifo#2\efifo}
        \expandafter\aux\fi
}%end \bfifo...\efifo
```

or via

```
\def\bfifo#1#2\efifo{\process{#1}
   \ifx\empty#2\empty\let\aux=\relax
   \else\def\aux{\bfifo#2\efifo}
   \fi\aux
}%end \bfifo...\efifo
```

A more TEX-like implementation is

```
\def\bfifo#1{%
   \ifx\efifo#1\let\nxt=\relax%
   \else\def\nxt{\process{#1}\bfifo}%
   \fi\nxt}%end \bfifo
```

The advantage of the last implementation is that the input stream is processed one group or token at a time until `\efifo` is encountered. Moreover, it can handle the invoke `\bfifo\efifo`, the empty case. No auxiliary stacks are involved. This way of programming is unusual for those familiar with PASCAL-like programming.

[1] See Knuth (1968), section 2.2.1.

[2] In the Tower of Hanoi article Knuth's list datastructure was finally used — TEXbook Appendix D.2 — with FIFO inherent.

[3] My first version had the two tokens after `\ifx` reversed — a cow flew by — and made me realize the non-commutativity of the arguments of TEX's conditionals. In math and in programming languages like PASCAL the equality relation is commutative! Note that at least one argument is needed in the above given implementation of FIFO.

[4] If a list has to be *created*, Knuth's list datastructure might be used, however, simplifying the execution of the list. See TEXbook Appendix D.2.

[5] Note that the second `\empty` is not always necessary. Knuth and Mackay (1987) demonstrate yet another variant of programming the test. The above given form is in agreement with Knuth's style as demonstrated in `\displaytest`, see the TEXbook, Appendix D-1, p.376.

[6] See Kabelschacht (1987).

2.1 Variable number of parameters

TEX macros can take at most 9 parameters. The above `\bfifo` macro can be seen as a macro which is relieved from that restriction. Every group or token in the input stream after `\bfifo` will become an argument to the macro. The first token or group is the first argument to the first invoke. This invoke ends with an invoke of itself using the next token or group from the input stream as argument. So the second token is argument of the second invoke. In general the n^{th} token or group is argument of the n^{th} invoke and so on until the `\efifo` token is reached, whereupon no invoke of `\bfifo` will occur.[7]

2.2 Length of string

An alternative to Knuth's macro, TB219, is obtained via an appropriate definition of `\process`.

```
\newcount\length
\def\process#1{\advance\length1}
```

Then `\bfifo aap\efifo` yields the length **3**.

2.3 Vertical printing

David Salomon treats the problem of vertical printing in his courseware. Via an appropriate definition of `\process` and a suitable invoke of `\bfifo..\efifo` it is easily obtained.

```
\def\process#1{\hbox{#1}}
xy\vbox{\bfifo abc\efifo}yx
```

yields xy a b yx.[8]

2.4 Delete last character of argument

Again an example due to David Salomon. It is related to the well-known `\gobble` macro to eat the *next* token (or group) from the input stream. One could define an appropriate `\process` but that will require double testing. Simpler is the following modification of the `\bfifo...\efifo` template.

```
\def\bgobblelast#1#2\egobblelast{
   \ifx\empty#2\empty\let\aux=\relax
   \else#1\def\aux{\bgobblelast#2%
    \egobblelast}\fi\aux%
}%end \bgobblelast...\egobblelast
```

Then `\bgobblelast aap\egobblelast` will yield `aa` .

2.5 To process words

In document preparation it is important to be able to handle quantities sequentially as elements of a list. What about handling a list of words? Amy Hendrickson in her courseware considers among others the problem of underlining words. This can be done by underlining every character, but that is slow. A faster solution can be obtained by first modifying the `\bfifo...\efifo` template into a version which picks up words, and to give `\process` the function to underline its parameter.[9]

```
\def\bfifow#1 #2\efifow{\process{#1}%
%Process words recursively;
%no addition of space here (part of
%\process if needed).
  \ifx\empty#2\empty\let\aux=\relax
  \else\def\aux{\bfifow#2\efifow}%
  \fi\aux}%end \bfifow...\efifow
```

The more TEX-like implementation, where the input stream is processed word wise, reads

```
\def\bfifow#1 {%
   \ifx\efifow#1\let\nxt\relax
   \else\def\nxt{\process{#1}\bfifow}
   \fi\nxt}%end \bfifow
```

2.5.1 Underlining words

In print it is unusual to emphasize words by underlining. Generally another font is used, see discussion of exercise 18.26 in the TEXbook. However, now and then people ask for (poor man's) underlining of words. The following `\process` definition underlines words picked up by `\bfifow...\efifow`.

```
\def\process#1{\vtop{\hbox{\strut#1}
              \hrule}\ }
\leavevmode\bfifow leentje leerde lotje
        lopen langs de lange lindenlaan
\efifow\unskip.
```

yields:[10]
leentje leerde lotje lopen langs de lange lindenlaan.

Note that underlining of complete sentences has to be considered separately if underlining punctuation marks is forbidden. (One possibility is to redefine `\process` such that the last symbol of its argument is inspected and appropriate action taken; another possibility is to use the general `\bfifo...\efifo` macro and suppress underlining for punctuation marks via appropriate programming of `\process`.)

3 Nested FIFO

One can nest the FIFO paradigma for example for processing lines word per word.[11] The template reads

```
\def\bfifol#1^^M#2\efifol{
  \bfifow#1\efifow
  \ifx\empty#2\empty\let\auxl=\relax
  \else\def\auxl{\bfifol#2\efifol}
  \fi\auxl}%end \bfifol...\efifol
```

with `\bfifow...\efifow` as defined above.

[7] Another way to circumvent the 9 parameters limitation is to associate names to the quatities to be used as parameters, let us say via def's, and use these quatities via their names in the macro. This is related to the so-called keyword parameter mechanism of command languages.

[8] Note the use of the `\hbox...` in process.

[9] Note that underlining inhibits hyphenation.

[10] `\unskip` is needed to undo the insertion of the last space.

[11] Or character per character, token per token, or group per group.

3.1 Natural data

Data for `\h(v)align` needs & and `\cr` marks. We can get plain TEX to insert an automatic `\cr` at each (natural) input line, TEXbook p.249. An extension of this is to get plain TEX to insert `\cs`-s, column separators, and `\rs`-s, row separators, and eventually to add `\lr`, last row, at the end, in natural data. For example prior to an invoke of `\halign`, one wants to get plain TEX to do the transformation

P*ON	⇒	P \cs * \cs O \cs N \rs
DEK*		D \cs E \cs K \cs * \lr

This can be done via adaptation of the above template along with an appropriate `\process` definition.

```
\let\ea=\expandafter
\newdimen\csize\csize=3ex
\def\rs{\cr}%generalization of row sep
\def\lr{\cr}%last row
\def\cs{&}%generalization of column sep
\catcode'*=13 \def*{%crossed out cell
\vrule width0.6\csize height0.5\csize %
depth0pt}%simple BLACK variant
%more pleasing is the following
%poor man's grey
\newbox\crs
\setbox\crs=\hbox to.6\csize{\leaders%
\hbox to.2ex{\hss\vrule height.5\csize
depth0pt\hss}\hfil}
\def*{\copy\crs}
%
\catcode'\^^M=13 \let^^M=\relax
%Pick up and processing of lines
\def\bfifol#1^^M#2\efifol{%
\process{#1}%
\ifx\empty#2\empty\def\auxl{\lr}%
\else\def\auxl{\rs\bfifol#2\efifol}%
\fi\auxl}%ensure end conditional before
         %inserting tabular mark up
\def\process{\bfifo#1\efifo}%
%
%Pick up etc of chars per line
\def\bfifo#1#2\efifo{#1%#1 back
\ifx\empty#2\empty\let\aux=\relax%
\else\cs%insert \cs
\def\aux{\bfifo#2\efifo}\fi%
\aux}%
```

To demonstrate that it wor—-hey it works!—ks

```
%%%%%%% data provision %%%%%%%
\def\data{%
P*ON
DEK*
}%
%%%%%%% data transform %%%%%%%
\ea\def\ea\data\ea{\ea%
\bfifol\data\efifol}%define transform
%%%%%%%   application  %%%%%%%
$$\vbox{\halign{&\hbox to\csize{%
\vrule height.8\csize width0pt
depth.2\csize\hfil#\hfil}\cr\data}}$$%
```

will yield

P ▮ O N
D E K ▮

As may be guessed from the layout the above came to mind when typesetting crosswords, while striving after the possibility to allow natural input, independent of `\halign` processing. Note that a weak form of the look ahead principle is implicitly applied as well.

4 LIFO

A modification of the `\bfifo...\efifo` macro—`\process{#1}` invoked at the end instead of at the beginning—will yield the Last-In-First-Out template. Of course LIFO can be applied to reversion 'on the flight,' without explicitly allocating auxiliary storage.[12]

```
\def\blifo#1#2\elifo{%
   \ifx\empty#2\empty\let\aux=\relax%
   \else\def\aux{\blifo#2\elifo}\fi%
   \aux\process{#1}%
}%end \blifo...\elifo
```

With the identity—`\def\process#1{#1}`—the template can be used for reversion. For example `\blifo aap\elifo` yields paa.

5 Further reading

Zalmstra and Rogers (1989), apply the FIFO technique to a list of figures — or floating bodies — in order to merge the list appropriately with the main vertical list in the output routine. This is beyond the scope of this paper.

6 Conclusion

In looking for a fundamental approach to process elements sequentially—not to confuse with list processing where the list is also built up, see TEXbook Appendix D.2—TEX templates for FIFO and LIFO, emerged.

The templates can be used for processing lines, words or characters. Also processing of words or characters per line can be handled via nested usage of the FIFO principle.

TEX's conditionals are non-commutative, while the similar mathematical and programming operations are.

From the application point of view the FIFO principle along with the look ahead mechanism is applied to molding natural data into representations required by subsequent TEX processing.

[12] Johannes Braams drew my attention to Knuth and MacKay (1987), which contained among others `\reflect...\tcelfer`. They compare #1 with `\empty`, which is nice. The invoke needs an extra token, `\empty` — a so-called sentinel, see Wirth (1976) — to be included before `\tcelfer`, however. (Knuth and Mackay hide this by another macro which invokes `\reflect...\empty\tcelfer`). My approach requires at least one argument, with the consequence that the empty case must be treated separately, or a sentinel must be appended after all.

References

[1] Hendrickson, A (priv. comm.)

[2] Kabelschacht, A (1987): `\expandafter` in conditionals; a generalization of plain's `\loop`. *TUGboat* 8, no. (2), 184–185.

[3] Knuth, D.E (1968): The Art of Computer Programming. 1. Fundamental Algorithms. Addison-Wesley.

[4] Knuth, D.E, P. Mackay (1987): Mixing right-to-left texts with left-to-right texts. *TUGboat* 7, no. (1), 14–25.

[5] Knuth, D.E (1984): The TEXbook. Addison-Wesley.

[6] Laan, C.G. van der (1992): Tower of Hanoi, revisited. *TUGboat* 13, no. (1), 91–94.

[7] Salomon, D (priv. comm.)

[8] Wirth, N (1976): Algorithms + Data Structures = Programs. Prentice-Hall.

[9] Zalmstra, J, D.F. Rogers (1989): A page make-up macro. *TUGboat* 10, no. (1), 73–81.